in *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA: Morgan Kaufmann

Generality and Difficulty in Genetic Programming: Evolving a Sort

Kenneth E. Kinnear, Jr.
Adaptive Computing Technology
62 Picnic Rd.
Boxboro, MA 01719 USA
kim.kinnear@adapt.com

Abstract

Genetic Programming is applied to the task of evolving general iterative sorting algorithms. A connection between size and generality was discovered. Adding inverse size to the fitness measure along with correctness not only decreases the size of the resulting evolved algorithms, but also dramatically increases their generality and thus the effectiveness of the evolution process. In addition, a variety of differing problem formulations are investigated and the relative probability of success for each is reported. An example of an evolved sort from each problem formulation is presented, and an initial attempt is made to understand the variations in difficulty resulting from these differing problem formulations.

1 Introduction

In order to further the application of Genetic Programming to evolution of complex algorithms, the work reported here explores the impact of differing problem formulations and fitness measures on the likelihood of evolving a general sorting algorithm on a given Genetic Programming run.

1.1 Genetic Programming

Genetic Programming derives from Genetic Algorithms, developed initially by John Holland [Holland 1975] and extended since then by many others. Genetic Algorithms (GA) often work on fixed length, linear representations of genetic material, and operate on this material with fitness proportionate selection, reproduction, crossover, and mutation [De Jong 1987, Goldberg 1989].

A number of researchers have experimented with GA's applied to variable length genetic representations. Grefenstette [Grefenstette 1989] and Schaffer [Schaffer 1984] contain notable examples. Bickel has applied GA's to tree structured genetic material [Bickel 1987].

John Koza has developed Genetic Programming (GP), using analogues of GA genetic operators to directly modify and evolve tree structured programs (typically LISP functions) [Koza 1989, 1990, 1992]. While GP doesn't mimic nature as closely as do GAs, it does offer the opportunity to directly evolve programs of unusual complexity, without having to define the structure or the size of the program in advance. Koza has shown in [Koza 1992] that GP can be effectively applied to an unusual variety of problem areas.

1.2 Sorting as a Problem for Program Evolution

The work described here takes as its problem that of evolving a program which will correctly sort any sequence of numbers into order based on increasing size — clearly an open–ended problem. As a problem for GP, evolving a sorting algorithm is sufficiently difficult to provide a reasonable laboratory for exploring the relative difficulty of differing problem formulations. In addition, evolving a sorting algorithm which is fully general is a problem which necessarily involves sampling in the fitness tests, since the goal is an algorithm which will sort any sequence of numbers presented to it.

Sorting, as described above, is best performed either with iteration or recursion. The iterative approach is examined in this work, and extends the work reported by Koza on iteration in [Koza 1992, Chap. 18].

W. Daniel Hillis has done some fascinating work evolving sorting networks and co-evolving the test sequences at the same time [Hillis 1991], and by evolving for both correctness and with some pressure for efficiency has come close to the best 16-input network ever discovered. His work includes a variable length genome as well as some interesting techniques for spatial and competitive partner selection for recombination. While related to the work described here, Hillis' work focused on efficiency for a specific type of sorting problem. In contrast, the work reported here explores the impact of differing fitness measures on the likelihood of evolving a general sorting algorithm on a given GP run.

The impact of the general GP environment and fitness test selection on evolving sorting algorithms was explored by the author in [Kinnear 1993].

Una-May O'Reilly and Franz Oppacher have applied GP to a similar sorting problem [O'Reilly 1992]. They discuss some differences in difficulty created by alternative primitive selections. The work described here extends theirs in this area and reports experimental results on additional problem formulations.

Success, in the context of this work, is evolution of at least one general sorting algorithm which will sort any sequence to which it is applied. Differing fitness measures as well as problem formulations are explored, with the intent to increase the likelihood that such a general sorting algorithm will be produced on a given run.

2 Experimental Context

2.1 How to Recognize Success?

When evolving a sorting algorithm, a problem presents itself. Frequently, an evolved algorithm which performs successfully all of the fitness tests used (a completely fit individual) will fall short of complete generality. Therefore, a particular run is not terminated with the appearance of a completely fit individual. Instead, additional testing is performed on every completely fit individual and the run is terminated as successful only when one individual passes these more stringent tests. A similar approach is reported in [Koza 1992, p.245], although there the additional tests consisted of additional real numbers within the range [-1,+1]. For sorting, the additional tests consist of many more and longer random sequences.

The fitness tests used to drive evolution consist of 15 tests, 5 fixed and 10 random with a maximum size of 30, with the random tests changed each generation. The additional tests for generality (used only to terminate the run) consist of 300 random tests of maximum length 40, and all 256 sequences of length 8 consisting only of 0 and 1.

While many of the completely fit individuals fail to pass the additional tests (usually only 60% to 80% pass), no algorithm that has passed all of the additional tests has ever been shown to fail on any sequence subsequently presented to it. Many of these evolved and potentially fully general algorithms have been tested with thousands of random sequences of lengths of up to 600 with not a single failure. Thus, while testing can never prove generality, the sorting problem as expressed with these primitives appears to have a certain threshold of testing that, once passed, ensures at least a very high likelihood of generality.

2.2 GP Environment

In this problem, the structures that undergo adaptation are LISP s-expressions, composed of functions and terminals described below. Their only useful results are their side effects — the function's result is discarded. The side effect desired is to reorder a sequence of integers so as to leave it in a "sorted" order, small to large.

2.2.1 Initial Creation of the Population

A population of 1000 individual function trees are randomly generated with a depth less than or equal to 6, generated by uniform selection over the list of functions (and terminals for the terminal nodes of the tree). A difference between this work and that reported in [Koza 1992] is that the initial population in this work was not guaranteed unique.

2.2.2 Steady State GP

Steady State GP (SSGP) was used, as developed by Craig Reynolds [Reynolds 1992] from work by Gilbert Syswerda on steady state genetic algorithms [Syswerda 1991].

Interest is increasing in steady state techniques for genetic algorithms. Holland originally discussed a variety of reproductive plans for genetic algorithms in [Holland 1975], including one that was essentially steady state. De Jong evaluated overlapping generations in genetic algorithms in [De Jong 1975] and determined that for small populations (< 100) the negative effects of genetic drift appeared to outweigh the benefits of the steady state approach. Steady state GA is also discussed in some depth in by Lawrence Davis in [Davis 1991], and an early description by Darrell Whitley appears in [Whitley 1989]. More recently De Jong has again examined the steady state approach from both a theoretical and empirical standpoint [De Jong 1993], and suggests that the differences between steady state and generational GA's lie more in the results of the different selection and deletion strategies used than in the difference in fitness evaluation strategies.

Briefly, steady state GP differs from generational GP (as described in [Koza 1992]) in that individuals are created one at a time, evaluated immediately for fitness, and then merged into the population in place of an existing low fitness individual, whereas in generational GP an entirely new population of individuals is created each generation, and then they are all evaluated for fitness. In both cases the population size remains constant. Steady state GP as defined by Reynolds and implemented here also guarantees the uniqueness of each individual in the population, thus maximizing diversity. This guarentee of uniqueness adds only a small amount to the time necessary to process an individual.

A comparison of steady state GP to generational GP for the sorting problem described here can be found in [Kinnear 1993].

Since strictly speaking there are no generations in steady state GP, a *generation equivalent* is defined to have passed when the number of new individuals that have been generated is equal to the population size – 1000 for the work reported here. Throughout this paper a generation should be taken to mean a generation equivalent.

2.2.3 Genetic Operators

The following genetic operators were used to generate the new individuals. In all cases where points are chosen inside of existing functions, they are chosen with a 90% probability of being a function point and a 10% probability of being a terminal point, after [Koza 1990].

- **Single Crossover**, where two individuals are selected with fitness proportional selection (FPS), and a point is chosen inside a copy of each. A single new individual is created by replacing the subtree selected in one individual with the subtree from the other individual. The resulting new individual is restricted from growing deeper than a depth of 10, and the crossover points will be rechosen until this restriction is satisfied.
- **Non-fitness Single Crossover**, just like single crossover, except that the two individuals are selected using uniform random selection over the entire population instead of FPS.
- **Mutation**, where an individual is selected by FPS and then an internal point in a copy of the individual is replaced by a randomly generated tree. The resulting individual is restricted from growing by more than 15% beyond its current size.
- **Hoist**, (a special case of single crossover) where a new individual is created by selecting a point inside a copy of an existing individual chosen by FPS, and elevating ("hoisting") it up to be the entire new individual.
- **Create**, (a special case of mutation) where an entirely new individual is created in the same way as the initial random generation.

These genetic operators were used in the following probabilistic mix; Single and Non-fitness Single Crossover (.35 each), Mutation, Hoist, Create (.10 each).

Let's briefly examine the motivation for these genetic operators. The combination of single crossover (essentially a standard genetic operator) and non-fitness single crossover (decidedly non-standard) was shown in [Kinnear 1993] to improve the likelihood of success for the sorting problem described here. Presumably it allows potentially valuable fragments of genetic material held within otherwise undistinguished individuals to come together over several crossover operations, thus allowing larger genetic steps to be made than would otherwise be possible.

Hoist and create, rather non-standard operators, were also introduced in [Kinnear 1993] as special cases of crossover and mutation, respectively, and are designed to increase the likelihood that the resulting individual will be smaller than its parent. This is in response to the tendency for the individuals in this problem to grow without bound. While the effective results of both hoist and create can occur as the result of the more standard genetic operators crossover and mutation, the likelihood of them actually occurring decreases as the individuals in the population grow – just when it is needed most. Thus, by creating these as explicit operators, the probability that they will occur is increased as well as held constant despite considerable growth in the average individual size.

2.2.4 Fitness Calculations

The basic fitness measure used is to present an individual with an "unsorted" sequence and then execute the function tree. The number of "inversions", a measure of the disorder of the sequence [Knuth 1973, p. 11], is calculated before and after the execution of the function, and the fitness is based on the disorder remaining after the execution of the function

Adjusted and normalized fitness were used as in Koza [Koza 1992]. For a population size of M, the adjusted fitness ***adj(i)*** and normalized fitness ***norm(i)*** are calculated for an individual ***i*** as follows.

$$adj(i) = \frac{1}{1 + raw(i)} \tag{1}$$

$$norm(i) = \frac{adj(i)}{\sum_{k=1}^{M} adj(k)} \tag{2}$$

The raw fitness, ***raw(i),*** was developed in several steps. First, if the remaining disorder for a test ***t***, ***rdis(t)*** was greater than the initial disorder, ***idis(t)***, then the additional disorder was multiplied by 100 in (3) to yield the penalty disorder, ***pdis(t)*** in order to severely penalize algorithms which sorted some sequences at the expense of others. Then, the result ***res(t)*** for each test ***t*** was calculated in (4) from the remaining disorder ***rdis(t)*** added to the penalty disorder, ***pdis(t).***

$$pdis(t) = max(rdis(t) - idis(t), 0) \times 100 \tag{3}$$

$$res(t) = rdis(t) + pdis(t) \tag{4}$$

Then, the provisional raw fitness ***praw(i)*** is calculated in (5) by summing the results of all of the tests ***res(t)*** and multiplying by an order factor ***of***. To this is added a size component where the size of the function is multiplied by a size factor, ***sf***. Finally, the actual raw fitness ***raw(i)*** is calculated in (6) by subtracting the minimum provisional raw fitness found in the entire population, ***minpraw.***

$$praw(i) = \left(\sum_{t=1}^{15} res(t) \right) \times of + size(i) \times sf \tag{5}$$

$$raw(i) = praw(i) - minpraw \tag{6}$$

Unless otherwise noted, both ***sf*** and ***of*** have a value of 5.

3 Evolving for Generality

For an open–ended problem such as sorting, no finite number of fitness tests can ever do more than suggest the direction for evolution of a fully general sorting algorithm. Additional tests for generality were employed as termination criteria in all of the runs described here, as discussed in Section 2.1. A curious connection appeared between the likelihood of a completely fit individual passing the additional tests for generality, and the generation at which the individual appeared. The later in the run the completely fit individual appeared, the less likely that it would pass the additional generality tests.

When casting about for why this might be so, it was noted that the later in the run a completely fit individual appeared, the larger it was. Thus, there was a possible connection between size and generality, with the generality being inversely proportional to size.

Early work in evolving sorting algorithms using GP demonstrated a strong tendency for the individuals in the population to grow without bound. Stringent depth checks were implemented to avoid them growing deeper, but they would still grow in size by growing wider, using certain functions simply as connectives. Indeed, all but a very few of the functions that tested as general were so large as to defy any human understanding of them, even after simplifying them in obvious ways. It was easy to believe that as the functions grew, it became less and less likely for them to be general — as they were frequently hopelessly opaque to human understanding.

The default value for the order factor ***of*** in the runs discussed above was 100. The default for the size factor ***sf*** was 0. In an attempt to apply some pressure to keep the size down, and perhaps evolve more general algorithms, ***sf*** was increased from 0 to 10. No significant difference was noted in the likelihood of evolving a general sort on a given run.

Then ***sf*** was increased beyond ***of***. This was not a particularly intuitive thing to do, since who wants small, understandable, incorrect sorting algorithms? The results were unexpected. Individual runs tended to take longer to produce a general individual, but 20 out of 20 runs did produce at least one general individual, a significant increase over the previous best comparable case of 7 out of 20!

When ***of*** and ***sf*** were set equal, both to 5, a highly repeatable set of successful runs was possible.

Several sets of 20 runs were made to further explore the space around the 5,5 pair, with ***sf*** held equal to 5 and ***of*** varied from 1 to 20.

of

Figure 1: Varying *of* with *sf*=5

Figure 1 illustrates the results, with three curves. The top curve shows the cumulative probability of success at generation 49, on the left Y axis. This shows that the more size is a factor (by reducing the order factor) the greater the likelihood of a particular run finding at least one general individual. The trade-off is that it will take longer to produce a general individual on the average, as shown by the bottom curve, which shows the average number of individuals required to produce the first general one, on the right Y axis. The middle curve shows the values from the bottom curve divided by the values from the top as a rough measure of efficiency. The valley from ***of***=3 to ***of***=7 shows the area of greatest payoff.

As with many results, this one poses as many questions as it answers. Are there other problems for which generality of the result is inversely proportional to the size of the resulting function? Clearly this is only an issue for problems where sampling is required for the fitness functions and thus generality is not assured by passing the fitness tests, but does it hold for many of these types of problems?

4 Primitive Selection and its Effect on Difficulty in GP

There are a number of ways that the primitive selection and definition can affect the difficulty that GP will have in solving a particular problem. These range from the particular ways that the functions interpret their arguments through the types and choices of return values of the functions, to different definitions of the functions themselves. This section examines the effect of differing primitive selections on difficulty.

4.1 Primitive Definitions

In order to apply GP to a problem, a list of primitives (functions and terminals) sufficient to solve the problem

must be created. Terminals are the constants or variables. The functions used for sorting break down into three classes; arithmetic functions, sequence comparison and manipulation functions, and an iteration function.

- ***Terminals***: Two terminals were used, `*len*` and `index`. `*len*` takes on the value of the number of elements in the sequence to be sorted. It is not a valid sequence index, however, as the sequences are indexed starting at zero. `index` is the iterator variable, which takes on a variety of values when used inside of the `work` of the iterator function, and 0 if used outside of the iterator.
- ***Arithmetic Functions***: The three arithmetic functions are `(e- n m)`, `(e1+ n)`, `(e1- n)`. These are protected versions of the standard Common LISP functions `-`, `1+`, and `1-`. They return `n-m`, `n+1`, and `n-1` respectively. They differ from the standard functions in returning 0 when the arguments are non–numeric instead of causing an error.
- ***Sequence Comparison and Manipulation Functions***: The real work of the sorting algorithm is performed by the side effects of these functions. In each case, `x` and `y` are indices into the sequence under test, and not the actual values present in the sequence. For all of these functions, if `x` and `y` are non–numeric or not valid indices into the sequence, the functions return 0.

  `(order x y)` and `(swap x y)` are similar, and one or the other must be used in every primitive set, as they are the only functions which actually change the order of the sequence. `order` will swap the sequence elements if the first is larger than the second. `swap` will always swap the sequence elements indexed by its two arguments, and requires other control structures to determine when it should be used.

  `(wismaller x y)` and `(wibigger x y)` take sequence indices and return the index of the sequence element of the smaller or larger, respectively.

  `(if-lt x y work)` is a conditional which executes its `work` only if the sequence value at index `x` is less than the sequence value at index `y`.

  `(less x y)` and `(if tst work)` are simply primitives which split the `if-lt` function into two more natural parts, and are used together. `less` returns 1 if the sequence value at index `x` is less than the sequence value at index `y`, and 0 in all other cases. `if` executes `work` if `tst` is non–zero.
- ***Iteration Function:*** `(dobl start end work)` takes a starting index value `start`, an ending index value `end`, and a `work` clause to execute. It sets `index` equal to the `start`, and increments `index` by 1 until either `end` or `*len*` is reached, executing `work` whenever the value of `index` is a valid sequence index into the sequence under test.

4.2 Difficulty Comparisons of Alternative Function Sets

This section examines the differences in difficulty created by markedly different function choices. Table 1 shows the differing sequence manipulation and comparison functions used by the four sets of runs. Figure 2 shows the results of 20 runs of each of these four problem formulations. The curves show the cumulative probability of success at each generation for each problem.

Figure 2: Comparisons of Difficulty

Clearly, `order` is the easiest primitive to use, as it requires only creating a doubly nested loop and passing `order` across the sequence n**2 times (for a sequence of length n). The combination of `swap` with `wismaller` and `wibigger` is next (called small), where no conditionals are required, but the correct numbers to swap just

.

Table 1: Set Definitions

Set Name	Sequence Manipulation Functions	Common to All Sets
order	`(order x y)`	`(dobl start end work)` `index` `(e1- x)` `(e1+ x)` `(e- x y)` `*len*`
small	`(wismaller x y)(wibigger x y)(swap x y)`	
if-lt	`(if-lt x y work)(swap x y)`	
if	`(if tst work)(less x y)(swap x y)`	

flow from the comparison functions to the `swap` function. The `if-lt` formulation with swap is a bit harder, for reasons that are not entirely clear. The traditional combination of `if` and `less` with `swap`, about what a human would choose as a reasonable set, is the hardest used in this particular experiment, and is considerably harder than any of the others.

4.3 A Gallery of Sorts

Let's examine some of the sorts that have evolved using each of the primitive sets in the last section, and then continue the discussion of relative difficulty. These are all taken from runs with ***sf***=5 and ***of***=5. The examples below were selected from among the simplest of the evolved sorts and contain only one type of expression that could be further simplified. This is the generation of a constant 0, and as an aid to understanding, all the expressions that evaluate to a constant 0 are shown in a strike-through font, ~~like this~~, below.

4.3.1 `order`

An evolved sort using the `order` primitive is shown in Figure 3. Since the expression `(e- index index)` evaluates to 0, it is shown in a strike-through font. There are few surprises here.

```
(dobl index
      *len*
      (dobl (e- index index)
            *len*
            (order (e1- index) index)))
```

Figure 3: Example of `order`

4.3.2 `swap wismaller wibigger`

An evolved sort using `swap`, `wismaller`, and `wibigger` is presented in Figure 4. This is a minimal completely correct solution to the problem, again with few surprises other than those that stem from the unusual primitives involved.

```
(dobl index
      *len*
      (dobl (swap index *len*)
            *len*
            (swap
              (wibigger (e1- index)
                        index)
              index)))
```

Figure 4: Example of `small`

4.3.3 `if-lt swap`

An evolved and completely correct sort using `if-lt` is shown in Figure 5.

```
(dobl index
      *len*
      (dobl (swap *len* *len*)
            *len*
            (if-lt index
                   (e1- index)
                   (swap
                     (e1- index) index))))
```

Figure 5: Example of `if-lt`

4.3.4 if less swap

Figure 6 shows a sort evolved using the `if`, `less`, and `swap` primitives — except that it doesn't use the `if` primitive at all! Surprisingly, the expected sort (shown in Figure 8) failed to evolve in any run with a population size of 1000.

```
(dobl
  index
  *len*
  (dobl
    (less index *len*)
    *len*
    (swap (less
            (e1+ (swap index
                       (less index
                             (swap *len*
                                   *len*)))
            (swap index *len*) )
           index)))
```

Figure 6: Example of `if`

Figure 7 shows another sort that was evolved from `if`, `less`, and `swap`. Again, it does not use `if` at all!

```
(dobl
  index
  *len*
  (swap index
        (dobl (swap *len* *len*)
              (e1- *len*)
              (swap index
                    (e- *len*
                        (less (e1- *len*)
                              index))))))
```

Figure 7: Example of `if`

Figure 8 shows the sort that one would expect from `if`, `less`, and `swap`. It evolved only once in a series of 20 runs with a population size of 4000.

```
(dobl
  index
  *len*
  (dobl (swap *len* index)
        *len*
        (if (less (e1+ index) index)
            (swap index
                  e1+ index)))))
```

Figure 8: Example of `if` (Population Size of 4000)

4.4 How to Characterize Difficulty?

In the preceding sections we have seen numeric evidence that the difficulty of the sorts shown varies rather widely. While some of the reasons for this are clear, as in the case that uses `order`, many of the other reasons are not.

One characterization for difficulty might be the minimum size of a completely correct sorting algorithm. Table 2 shows the minimum correct size (from the examples in Section 4.3) and the average minimum size for general sorts produced over a series of 20 runs. This would explain in general terms the difficulty difference between the first three primitive styles, shown below, but is inadequate to explain the preference for avoiding `if` in the `if`, `less`, and `swap` case. In all cases, the sorts evolved without `if` are longer than the sort that does use `if`.

Table 2: Size Comparisons

Set Name	Fig.	Size (†⇒minimum correct size)	Avg. Min Size
order	3	12†	14.20
small	4	14†	16.94
if-lt	5	16†	29.53
if	6	22	23.43
	7	19	
	8	17†	n/a

Note also the relationship between the difficulty shown in Figure 2 and the pattern of sizes for the order, small, and if-lt cases – and how this breaks down for the if case.

5 Conclusions/Summary

- Including size as well as sorting ability in the fitness calculations not only has an effect on the size of the resulting algorithms, but also increases the likelihood of evolving a general sort. This supports results reported by Koza in [Koza 1992] which shows that multiple fitness measures can be effective, and shows as well that they can have additional and potentially positive effects.
- The way that the primitives are defined for a problem in GP has a large effect on the difficulty of solving the problem. Primitives for sorting can be defined in a variety of ways, and four differing problem formulations show a wide range of relative difficulties. An evolved sort from each formulation was examined yielding one non-intuitive result – a preference for avoiding the `if` construct when it was offered. While the minimum size correct algorithm is clearly a partial determiner of difficulty, other factors are involved which are not obvious.

6 Further Work

One major open question is whether simple GP can be further "tuned" with relatively small changes to the environment and parameters to support evolution of considerably more complex sorting algorithms as a step toward allowing evolution of other complex algorithms. The `if`, `less`, `swap` formulation above appears to have about pushed this GP environment to its limit. What changes are necessary to allow evolution of even more complex sorting algorithms? Parameterized subroutines (functions) as developed by Peter Angeline [Angeline 1992] have been tried, without any initial success. They would likely help evolve solutions to more complex problems if the increase in complexity included the potential for significant reuse of common substructures, which in these experiments it does not.

An interesting question is whether the connection between high evolutionary pressures against increasing size and algorithms of greater generality holds for other problems.

Still another fascinating (and obvious) question concerns the result of adding efficiency to the fitness function, and attempting to evolve an efficient sorting algorithm.

7 Acknowledgments

Continued thanks to John Koza for support and encouragement, as well as for defining GP and the amazing amount of work that went into his book. The ICGA reviewers gave unusually perceptive and helpful comments. Special thanks to my wife Karen for editorial reviews and exceptional support.

All of the code for this system was developed by the author on a 386 laptop using XLISP-PLUS 2.1d by Tom Almy and David Betz. The experiments were run under Lucid Common LISP.

8 Bibliography

Angeline, P. J. (1992) and J. B. Pollack, "Coevolving High-Level Representations." LAIR Technical Report 92-PA-COEVOLVE, The Ohio State University, Columbus, OH. Submitted to *Artificial Life III*, C. G. Langton, Ed.

Bickel, A. S. (1987) and R. W. Bickel, "Tree Structured Rules in Genetic Algorithms," in *Proceedings of the 2nd International Conference on Genetic Algorithms*, J. Grefenstette, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates.

Davis, L. (1991) *Handbook of Genetic Algorithms*. New York, NY: Van Nostrand Reinhold.

De Jong, K. A. (1975) *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Doctoral Thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI.

De Jong, K. A. (1987) "On Using Genetic Algorithms to Search Program Spaces," in *Proceedings of the 2nd International Conference on Genetic Algorithms*, J. Grefenstette, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates.

De Jong, K. A. (1993) "Generation Gaps Revisited," in *Foundations of Genetic Algorithms, 2*, L. D. Whitley, Ed. San Mateo, CA: Morgan Kaufmann.

Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Grefenstette, J. J. (1989) "A System for Learning Control Strategies with Genetic Algorithms," in *Proceedings of the 3rd International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. San Mateo, CA: Morgan Kaufmann.

Hillis, W. D. (1991) "Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure," in *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer and S. Rasmussen, Eds. Reading, MA: Addison-Wesley.

Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: The University of Michigan Press.

Kinnear, K. E. Jr. (1993) "Evolving a Sort: Lessons in Genetic Programming," in *Proceedings of the 1993 International Conference on Neural Networks*, New York, NY: IEEE Press.

Knuth, D. E. (1973) *Sorting and Searching*. Reading, MA: Addison-Wesley.

Koza, J. R. (1989) "Hierarchical Genetic Algorithms Operating on Populations of Computer Programs," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann

Koza, J. R. (1990) "Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems." *Technical Report No. STAN-CS-90-1314*, Computer Science Department, Stanford University.

Koza, J. R. (1992) *Genetic Programming*. Cambridge, MA: MIT Press.

O'Reilly, U. M. (1992) and F. Oppacher, "An Experimental Perspective on Genetic Programming," in *Parallel Problem Solving from Nature, 2*, R. Manner and B. Manderick, Eds. Amsterdam, The Netherlands: Elsevier.

Reynolds, C. W. (1993) "An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion," in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, J. A. Meyer, H. L. Roitblat, and S. W. Wilson, Eds. Cambridge, MA: MIT Press.

Schaffer, J. D. (1984) *Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms*. Doctoral Dissertation, Department of Electrical and Biomedical Engineering, Vanderbilt University, Nashville TN.

Syswerda, G. (1991) "A Study of Reproduction in Generational and Steady-State Genetic Algorithms.", in *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. San Mateo, CA: Morgan Kaufmann, 1991.

Whitley, D. (1989) "The GENITOR Algorithm and Selection Pressure: Why Rank-Based allocation of Reproductive Trials is Best." in *Proceedings of the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. San Mateo, CA: Morgan Kaufmann.